PL-TR-93-2264

CMCAM: A HIGH PERFORMANCE CM-5 LATTICE GAS SIMULATOR

G. P. Seeley

Radex, Inc.
Three Preston Court
Bedford, MA 01730

December 30, 1993

Scientific Report No. 4

PHILLIPS LABORATORY
Directorate of Geophysics
AIR FORCE MATERIEL COMMAND
HANSCOM AIR FORCE BASE, MA 01731-3010

"This technical report has been reviewed and is approved for publication"

EDWARD C. ROBINSON
Contract Manager
Data Analysis Division

ROBERT E. McINERNEY, Director
Data Analysis Division

This report has been reviewed by the ESD Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS).

Qualified requestors may obtain additional copies from the Defense Technical Information Center. All others should apply to the National Technical Information Service.

If your address has changed, or if you wish to be removed from the mailing list, or if the addressee is no longer employed by your organization, please notify PL/TSI, 29 Randolph Road, Hanscom AFB, MA 01731-3010. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	30 December 1993	Scientific Report No. 4

4. TITLE AND SUBTITLE

CMCAM: A High Performance CM-5 Lattice Gas Simulator

5. FUNDING NUMBERS

PE 62101F
PR 7659 TA GY WU AA

Contract F19628-93-C-0023

6. AUTHOR(S)

G. P. Seeley

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

RADEX, Inc.
Three Preston Court
Bedford, MA 01730

8. PERFORMING ORGANIZATION REPORT NUMBER

RXR-93121

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Phillips Laboratory
29 Randolph Road
Hanscom AFB, MA 01731-3010

Contract Manager: Edward C. Robinson/GPD

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

PL-TR-93-2264

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for Public Release
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

CMCAM, an extendable lattice-gas simulation program for the Thinking Machines Corporation CM-5 supercomputer is described. CMCAM manipulates the vector processing accelerator units present on each CM-5 processing node using GCC/DPEAC, a macro package for issuing assembler instructions from a C program. The system is able to run several lattice gas rules and extract time and space averaged data for further processing. X-Window graphics are built into the code for remote display of 2-d hydrodynamic flows. An example case of 2-d flow past a flat plate obstacle is described. Performance results show that lattice gas simulations can be run at update rates of 550 million sites/second on a 256 node partition. This report contains detailed information on the implementation methodology used on the CM-5 and may be useful for persons interested in implementing other high-performance computations on the CM-5 that use the vector units directly.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Lattice gas automata, Hydrodynamics, Parallel supercomputing	24
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Jeff Yepez for his interest in and support of this project. I would also like to acknowledge the contribution of Pablo Tamayo of TMC/Los Alamos National Laboratories who got me pointed in the right direction at the very beginning. The Free Software Foundation provided the GCC compiler that dramatically shortened the assembler coding time. The support staff of the AHPCRC in Minnestota was always helpful and responded quickly to both real and imagined problems. The color illustration was done using IMSL/IDL.

1. INTRODUCTION

This report describes CMCAM, a program for performing lattice-gas calculations using the Thinking Machines Corporation CM-5 supercomputer. CMCAM represents part of a continuing investigation into the applicability and performance of contemporary parallel supercomputers for lattice gas simulation. The development of this code was guided by two goals. The first goal was to achieve high performance in order to address significantly larger problems than can be handled with ordinary high performance workstations. The second goal was to produce a code that is easy to modify for new experimental scenarios. An earlier implementation done using only high level language facilities was deemed too slow to be really useful. The present implementation uses assembler language for the highest possible performance. The use of an assembler language certainly increases the complexity of the code, but provides an enormous boost in the delivered performance. The code has been designed with the hope that future extensions and enhancements to the code will involve a minimum amount of assembly code manipulation. The code has been built with an eye towards compatibility with another parallel supercomputer architecture, the MIT Information Mechanics group's CAM-8 machine [*Margolus*, 1993]. Many components of a CMCAM simulation can be re-used without change on the CAM-8. After reading this report an experienced C programmer should be able to understand the techniques used to perform lattice gas simulation on the CM-5. The report should also aid persons interested in modifying the code for future lattice gas scenarios.

2. LATTICE GAS MODELING

Lattice-Gas-Automata (LGA) models represent an intriguing alternative to conventional methods of hydrodynamic simulation. These exactly computable models are based on particles moving on a uniform lattice with discrete velocities. Particles are typically represented using individual bits to indicate their presence or absence at a particular site. The particles interact with each other through collisions that conserve the desired invariant quantities, typically mass, momentum and energy. The collisions contain the physics of the particular system under study.

An LGA time step can be separated into two phases. The first phase involves streaming of the particles to their new locations, consistent with their velocity and the lattice on which the simulation is being performed. Once all the particles are at the appropriate lattice sites they interact, according to the specified "rules" of the simulation. After the collision process the streaming step is repeated.

LGA are numerically stable methods that are able to easily accommodate highly irregular boundary conditions. The FHP model developed by Frisch Hasslacher and Pomeau rigorously shows 2-D Navier Stokes flow in the incompressible limit [*Frisch, et al.*, 1986]. LGA modeling methodology has been extended to 3 dimensional hydrodynamic flows with the advent of the FCHC model [*d'Humieres, et al.*, 1986]. Due to the uniformity and concurrency of the lattice gas update process, implementation on parallel computers is usually efficient, requiring only local communication. LGA modeling offers many substantial advantages over conventional finite-difference techniques and it is attracting increasing attention as a promising new approach to fluid flow [*Doolen*, 1990].

3. CM-5 ARCHITECTURE

The Thinking Machines Corporations's CM-5 is a massively parallel computer that can contain up to 16384 processing nodes [*Thinking Machines Corp.*, 1992]. Figure 1 shows an individual processing node consisting of a SPARC CPU, 32 Mbytes of memory and 4 Vector processing units.

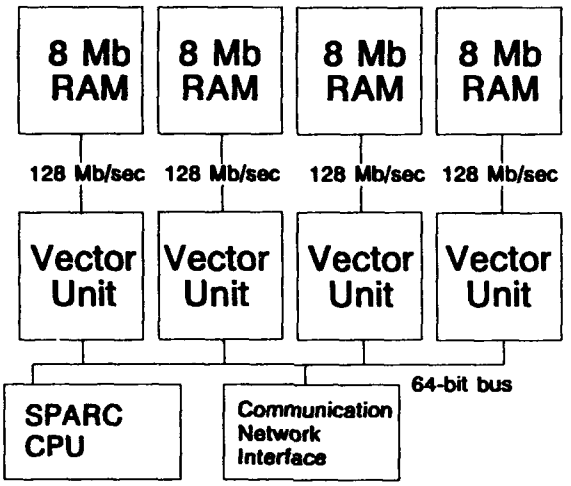

Figure 1. CM-5 Node Architecture

These processing nodes are all connected via a "fat-tree" communications network that allows fast inter-node communication. These processing nodes are controlled by a front-end host computer which is a modified SUN workstation. The SPARC processor on each node issues instructions to the vector units and performs most address bookkeeping tasks while the vector units perform arithmetic and logical operations on the data. Each vector unit has a peak rate of 32 million 64-bit ops (floating point or integer) for a combined total of 128 Mops/node. Each node's memory is divided into 8 Mbyte banks, one for each vector unit. The banks of memory are mapped into distinct parts of the address space, inter-bank communication is be mediated by the SPARC processor in most cases. Each vector unit has it's own independent 128 Mbyte/sec path to memory for a combined memory bandwidth of 512 Mbyte/sec for each node. The vector units also act as high performance memory interfaces when their arithmetic and logical capabilities are not being used. The CM-5 at the Army High Performance Computing Research Center in Minneapolis, Minnesota currently contains 512 nodes for a total of 16 Gb of memory and 64 Gops of peak processing speed. A CM-5 at Los Alamos National Labs contains 1024 nodes, for twice the capacity.

CMCAM is implemented on the CM-5 in a Multiple Instruction Multiple Data (MIMD) style. The CMMD message passing library is used for inter-node communication and host-node interaction [*Thinking Machines Corp.*, 1993a]. In order to get the highest possible performance the vector units on each node are explicitly manipulated using their assembler language known as DPEAC. To ease the burden of hand coding the vector units a macro package known as GCC/DPEAC is used [*Thinking Machines Corp.*, 1993b]. This package uses features available in the GNU C compiler to issue assembler language instructions from ANSI C and simplifies matters considerably.

We partition the problem space into equally sized rectangular units. Figure 2 shows this partitioning for N nodes. Each processing node is responsible for updating one of these rectangular units. This partitioning allows one to send a small number of long messages to connect the space together. Inter-node communication is only necessary along one of the axes of the problem space. Since the inter-node communications network is optimized for long message lengths we expect that this partitioning will make effective use of available communications bandwidth. This approach also substantially reduces code complexity. Within a processing node, each of the 4 vector units is responsible for updating it's quarter of the space. Communication between each vector unit's 8 Mbyte bank of memory is mediated by the SPARC processor.

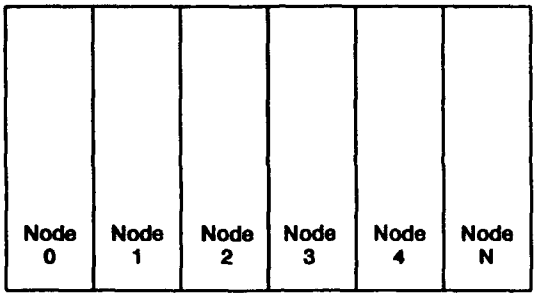

Figure 2. Problem Space Partitioning

There are two distinct phases of a lattice gas update cycle. The first phase is the collision phase where particles interact, this amounts to a local permutation of the data at a particular site. The second phase involves streaming of the bits to their new locations, consistent with their velocity and the lattice on which the simulation is being performed. In most lattice gas models all collisions can happen concurrently and all sites can stream their data concurrently, as well.

4. STRUCTURE OF A CM5 APPLICATION

There are several different paradigms for implementation of a particular application on the CM5. There are languages such as C* and CM-FORTRAN that insulate the implementor from explicit management of processors and allow the use of high level concepts and structures in a parallel environment. C* and CM-FORTRAN have a Single Instruction Multiple Data (SIMD) approach to parallelism. Typically there is a substantial performance loss when using these languages, as the compilers introduce substantial overhead in their management of the available resources.

There is another approach to parallel computing known as Multiple Instruction Multiple Data (MIMD). This model contains several processors, each executing it's own stream of instructions that communicate with other processors via message passing. This type of model is available through the use of the CM5 message passing library known as CMMD. CMMD provides communication and synchronization primitives that can be used to produce a MIMD application. These primitives can be manipulated from a standard high-level language such as ANSI C.

For many types of problems this explicit management of the communications resources can be more efficient than implementation in the SIMD high level languages. Lattice gas simulation tests have been performed with the CMCAM code that show a speed gain of a factor of 25 over a C* implementation [*Yepez, et al.*, 1994]. Since a main goal of a lattice gas implementation on the CM5 has been to produce an application with extremely high performance, the CMMD message passing model has been chosen and will be discussed here. The discussion here is based on Thinking Machines' CMMD documentation [*Thinking Machines Corp.*, 1993a].

There are two styles for programming a CM-5 application using the facilities provided by CMMD. There is a hostless and a host/node style. The host/node style has been used here. In a host/node type application there are two separate communicating main programs. There is a program that runs on the front-end host machine and a second program that is replicated on all the processing nodes. Since CMCAM performs such tasks as remote file transfer and X-window display the host/node paradigm is the most appropriate. The X-window code and all the disk I/O can be centralized in the program that runs on the front-end host.

The code is divided into several types of modules. Modules with a .cp.c extension are ANSI C program modules that run on the front end host. These modules are typically concerned with disk I/O, X-Window machinations or supplying/gathering simulation data to/from node code. Modules with a .pn.c designation are modules that make up the program that runs on each processing node. Modules designated .cdp.c contain GCC/DPEAC statements that can be directly mapped to VU assembly language instructions for the nodes.

A Makefile handles the compilation and linking tasks for the many types of modules. Briefly, the .cp.c modules are compiled/linked into a host program while the .pn.c modules are compiled/linked into a node program. The .cdp files are first compiled using gcc with flags that reserve some registers for exclusive use of the vector units. This initial compilation produces a file that can be run through dpas, the VU assembler. The output of dpas is a .pn.o object module that can be linked with the node code. After linking with uncountable libraries, the host and node programs are forged into a common executable that can be handled by the CM-5 run time system. Figure 3 shows the header file inheritance scheme for the host and node code. This scheme effectively partitions the definitions and declarations on both the host and node sides of the code, while allowing for some definitions to be commonly shared.

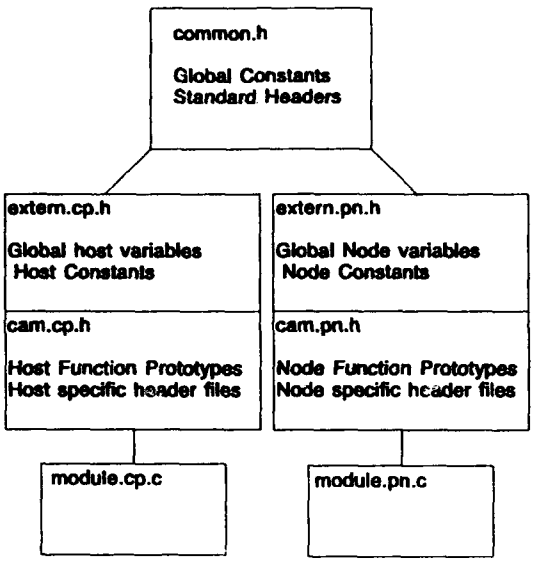

Figure 3. Header File Include Scheme

5. STREAMING

The most time consuming part of the code is the streaming. Even though each cell only communicates with its near neighbors during an LGA update cycle the communications part of the code is substantial, as individual bits must be picked out of words and reassembled into other words. In this CM-5 implementation there are two types of boundaries that the particles must move across. Particles must move across VU boundaries, since each VU has its own bank of memory. Particles must also move between processing nodes in the machine.

The streaming completely defines the lattice structure of the simulation. The first type of boundary is between each vector unit on a CM-5 processing node. Communication between the VU's is mediated by the SPARC processor. Values are read from the registers on each VU, transferred to a SPARC register and then written to the appropriate VU. This transfer takes place at each streaming step of the calculation and is accomplished using the **dpread** and **dpload** CDPEAC macros.

At the edge of each processing node there is a different boundary that must be crossed, this boundary is between two processing nodes. Communication between nodes involves the data network and is done using the CMMD message passing library calls. This type of communication is substantially slower than the on-node communication. For VU 0 and VU 3 there is an additional step of moving the data from the VU address space into the SPARC memory address space. The communication must ensure that every site on every VU has a directly accessible copy of the site data at each of its neighboring sites.

The process of updating each site after the communication process goes as follows:

1. Each site loads pointers to its neighbors from pre-computed addressing tables. These tables may be computed in ordinary C, which is advantageous for changing from one model to another. Additionally, potentially complex addressing calculations are performed only once, during initialization.
2. The pointers are dereferenced using the indirect addressing capability of the vector units.
3. Each site masks off the appropriate bits from it's neighbors and accumulates these bits in a register.
4. After all the bits from all the directions have been accumulated the value is written to memory. Memory is double buffered so that only old values are used in the composition of new ones.

Here is an actual DPEAC code fragment that performs these operations:

```
loadv_v_u(du,vlen,R0_addr,8,V6[0]); /* load pointers to R0 neighbors */
R0_addr += R_bump;                  /* bump pointer table address */
loadv_i(u,Sv,V6,V8);                /* load R0 neighbor data */
loadv_i(u,Sv,V7,V9);                /* load R0 neighbor data */
andv(du,V8,SCALAR(R126),V2);        /* R0 neighbor & SCALAR(R126) */

loadv_v_u(du,vlen,R1_addr,8,V6[0]); /* load pointers to R1 neighbors */
R0_addr += R_bump;                  /* bump pointer table address */
loadv_i(u,Sv,V6,V8);                /* load R1 neighbor data */
loadv_i(u,Sv,V7,V9);                /* load R1 neighbor data */
andv(du,V8,SCALAR(R126),V4);        /* R1 neighbor & SCALAR(R124) */

addv(du,V2,V4,V2);                  /* accumulate R1 bits */

storev_v_u(du,vlen,SNEXT_addr,8,V2[0]); /* write out accumulated results */
```

The first load instruction loads 2 32-bit pointers into each element of vector unit register V6, since it is a double unsigned (du) format instruction. Loading 2 single precision quantities as one double precision quantity doubles the effective memory bandwidth. These two 32-bit pointers show up as single precision quantities in V6 and V7. The addresses in V6 and V7 are then used as offsets to load the actual site data from neighbors in the R0 direction into V8 and V9. There are two indirect load instructions since we only want to load unsigned single precision (u) quantities. Then the bits from the neighboring site are subjected to a logical and operation which selects the bits that stream to the present site. Double precision masks have already been loaded into registers R126,R124 for this purpose. This process is repeated for all the lattice directions and the incoming bits are accumulated in a register using an add operation. After all the streaming has been completed the final accumulated results are written to memory as a double precision quantity.

6. COLLISIONS

The collision phase can be handled via look up tables (LUT's) for 16 bit sites. The LUT is attractive in that it can be an extremely simple and fast update mechanism. We have distributed the LUTs throughout the machine, indeed each vector unit has it's own copy of the LUT. Figure 4 shows the memory layout on each node. During the collision phase each vector unit fetches all the sites in it's partition of the problem space and runs them through its copy of the LUT. Since each vector unit has it's own independent 128 Mbyte/sec data path to a bank of memory, this operation can be performed extremely

rapidly. With this high degree of parallelism the LUT operation consumes a small fraction of the time necessary to update the space. As the number of bits of site data grows beyond 16 (64K entries), the LUT's begin to consume too much memory. For models that involve larger quantities of site data (i.e. \# bits $>$ 20) other methods involving LUT compression/decompression need to be used for the collision phase [*Henon*, 1992].

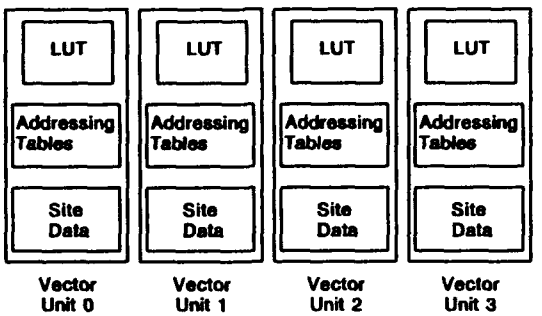

Figure 4. Node Memory Layout

Here is a GCC/DPEAC code fragment that performs collisions. This code assumes that we have loaded V2 with sites that are ready for collisions.

```
mulv(u,V2,SCALAR(R111),V6); /* multiply new state by 4 to use */
                            /* as byte offset into lookup table*/

loadv_i(u,clutv,V6,V8);     /* perform indirect load from lookup table */
```

The code fragment shows that only a multiply and an indirect load are necessary to perform a lookup table based collision.

7. AN EXAMPLE CALCULATION

Here is an example of how to conduct a particular simulation experiment using CMCAM. This calculation will utilize an 8 bit variant of the FHP model, which contains a rest particle and obstacle bit. The bit definitions are contained in **fhp_hood.h**. Initial conditions, boundary conditions and any required forcing for the flow need to be specified.

Let's consider a specific flow experiment in some detail. The flow to be examined is channel flow with a flat plate obstacle. Figure 5 shows a diagram of this situation. The first consideration is that we need a steady flow directed towards the right of the diagram. Viscous dissipation will reduce the flow velocity to zero unless we include some type of forcing which will keep the fluid moving down the channel. Another important aspect of performing this channel flow experiment is what to do about the inflow and outflow boundary conditions. If a simple periodic geometry is used, disturbances propagating along the channel could re-enter it and eventually dominate the simulation behavior.

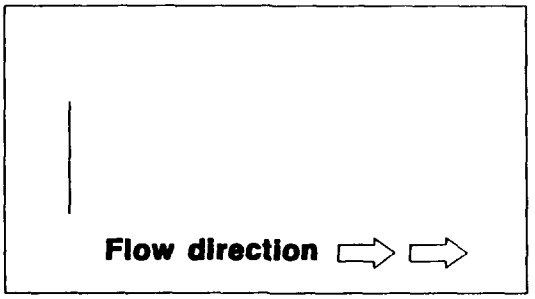

Figure 5. Flat plate channel flow.

A periodic channel geometry that incorporates a forcing strip located at the leftmost side of the channel meets the requirements. The forcing strip completely reconstructs the particle distribution in the strip at each time step. This prevents disturbances from propagating repeatedly around the channel. The reconstruction maintains the desired net velocity and density in the strip. Particles that emanate from the strip are free to propagate in either direction around the cylindrical channel.

Initial conditions may be constructed in two ways. Constructing a pattern file via an external C program is one way. This is good for running relatively small simulations. It allows easy reproduction of many experiments with a simple external program that needs only to know the required final output format. The second method is to construct a C routine for CMCAM that will construct the initial conditions at runtime. This method is good for performing extremely large simulations, where the initial pattern data might be too large to conveniently read in. The usual place to insert such a routine is in **init.pn.c**.

Let's examine the code for the initial conditions. This subroutine will first fill the simulation space with a fluid at a filling fraction of 1/7 with a net velocity of 0.4 momentum units per site in the positive x direction.

```
void init_plate(int li, int lj, float speed)
{
  int i,j;
                                /* initializes flat plate */
                                /* experiment at a filling fraction of 1/7 */

              /* all nodes execute this code */
              /* loop over the entire space on a node */
  for(i = 0; i < li; ++i){
     for(j = 0; j < lj; ++j){
                                /* put rest particles everywhere */
        PIPS(i,j,BIT6,Sv);

                                /* in 10% of cases replace 2 rest */
                                /* particles with two oppositely */
                                /* directed particles */
```

```
        if(zran1(0) < 0.1){
              /* avoid giving particles a negative x coordinate */
          if(j > 0){
            if(zran1(0) < 0.5){
              PIPS(i,j,BIT1,Sv);
              PIPS(i,j-1,BIT4,Sv);
            }
            else{
              PIPS(i,j,BIT0,Sv);
              PIPS(i,j-1,BIT3,Sv);
            }
          }
        }
                              /* insert particles with 1 unit of */
                              /* forward momentum to obtain an average */
                              /* momentum per site of [speed] */
        if(zran1(0) < speed){
          if(zran1(0) < 0.5){
            PIPS(i,j,BIT3,Sv);
          }
          else{
            PIPS(i,j,BIT4,Sv);
          }
        }
      }
    }

                                  /* insert bits for plate*/
    if(self_address == (partition_size / 8) ){
              /* this code only executes on one node */
       for(j = 0; j < 5; ++j){
              for(i = li/2 - li/16; i < li/2 + li/16; ++i){
                PIPS(i,j,BIT7,Sv);
              }
       }
    }

                              /* add hard walls at top and bottom of channel */
  for(j = 0; j < lj; ++j){
    i = 0;
    PIPS(i,j,BIT7,Sv);
    i = li - 1;
    PIPS(i,j,BIT7,Sv);
  }
}
```

An important thing to keep in mind is that this code runs on every processing node involved in the simulation. The loop boundaries li and lj denote the boundaries of each processing node's piece of the simulation space. The PIPS macro is used to load the values into the memory of each of the 4 vector units on each processing node. Note the self address conditional test where the flat plate obstacle is inserted. Each processing node has a unique variable called self address. The code that inserts the flat plate obstacle is only run on the node with a specific address. This test insures that the flat plate is inserted at only one point in the channel.

Now that we have specified the initial conditions let's examine the forcing scheme that keeps the fluid moving through the channel. As was previously discussed we will use a forcing scheme that not only forces the fluid, but also acts to prevent disturbances from recirculating around the channel. The code is similar to the initial conditions routine described above and would usually reside in **forcing.pn.c**.

```
void force_plate(float speed, int li, int lj)
{
  int i,j;
  int val;
  int r;
                          /* this forcing reconstructs */
                          /* the distribution at the forcing strip */
                          /* each time step */
                          /* assumes 1/7 filling fraction */

                          /* forcing strip in node 0 */
                          /* forces flow in positive x */

  if(self_address == 0){
    for(i = 0; i < li; ++i){
                          /* first lay down isotropic background */
                          /* choose a random bit to set */
      r = (int)(zran1(0) * 7.0);
      val = 1<<r;
                               /* put resulting value into vector memory */
    PIPS(i,2,val,Sv);
  }

    for(i = 0; i < li; ++i){
      if(zran1(0) < speed ){    /* put in +x speed particles */
                                /* to get appropriate speed */
        if(zran1(0) < 0.5 ){
          PIPS(i,2,BIT3,Sv);
        }
        else{
          PIPS(i,2,BIT4,Sv);
        }
      }
    }
  }
}
```

There is a self address test that makes sure the forcing is only done on one node. The rest of the code merely reconstructs a strip of fluid with a given net velocity. This routine is called after each time step to maintain a steady flow.

To compute the flow velocity at each point we use pre-computed lookup tables that map the individual site states to floating point numbers that specify density, x momentum, y momentum and so on. These tables are computed in **tabulate_states.pn.c**.

Nearly all the simulation ingredients are in place, we have completely specified boundary conditions and external forcing. To visualize the flow we need to extract the simulation data from each vector unit's memory, compute the flow velocity at each point and transmit an image to the front end host, where it can be displayed or written to disk for later processing. Figure 6 shows some results from running the experiment on a 1 K x 2 K lattice. The coloring indicates only the direction of the flow.

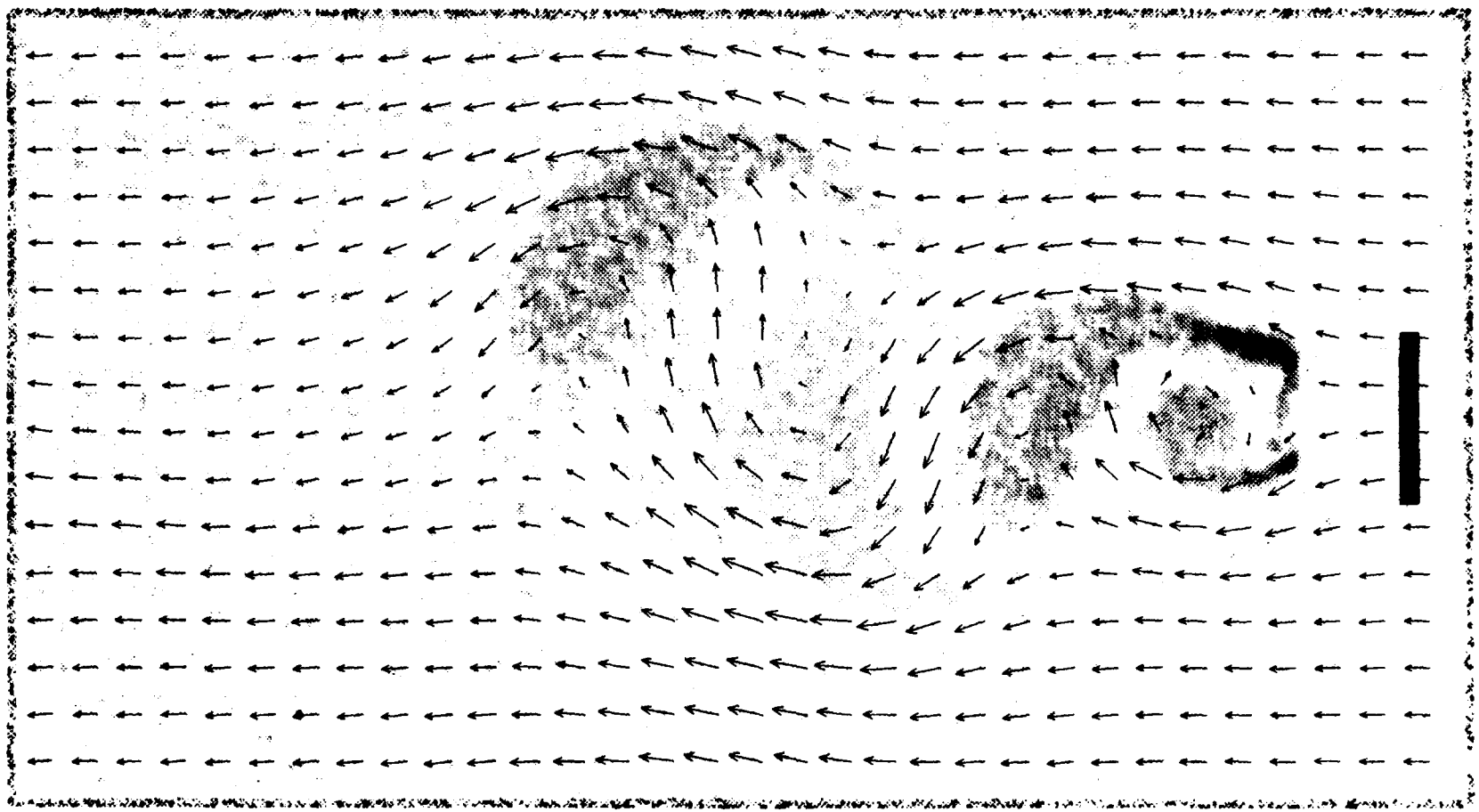

Figure 6. A Von Karman Vortex Street behind a flat plate.

8. PERFORMANCE

After implementing a simulator with the above considerations in mind we find that we can achieve update rates on the order of 550 Msites/sec on a 256 node partition of the CM-5 for an 8-bit FHP gas model. This timing was done using a 2048 x 32768 lattice with a model that packed two 8-bit sites into a 32 bit word. We find that the longer the system is across each node the greater the performance realized. This is due to the fact that long system sizes across each node increase the fraction of sites in the interior of each vector unit that do not need to communicate with sites on adjacent vector units or processing nodes.

Extending the CMCAM simulator to use more complex models can result in substantial performance penalties. For example the 2-speed thermohydrodynamic model of *Chen, et al.* [1991] requires that some bits are streamed twice as fast as other bits. The most straightforward way to implement this using CMCAM is to perform two streaming cycles before a collision cycle, incurring a factor of 2 performance loss.

REFERENCES

Chen, S., H. Chen, G. D. Doolen, S. Gutman, and M. Lee, "A Lattice Gas Model for Thermohydrodynamics", *J. Stat. Phys.*, 62(5/6):1121-1151, 1991.

d'Humieres, D., P. Lallemand, and U. Frisch, "Lattice Gas Models for 3D Hydrodynamics", *Europhys. Lett.*, 2:291, 1986.

Doolen, G. D (Ed.), *Lattice Gas Methods for Partial Differential Equations*, Addison-Wesley Publishing Company, 1990.

Frisch, U., B. Hasslacher, and Y. Pomeau, "Lattice Gas Automata for the Navier Stokes Equation", *Phy. Res. Lett*, 56:1505, 1986.

Henon, M., "Implementation of the FCHC Lattice Gas Model on the Connection Machine", *J. Stat. Phys.*, 68:353, 1992.

Margolus, N., "Cam-8: A computer architecture based on cellular automata", *Proceedings of the Pattern Formation and Lattice-Gas Automata Conference*, Fields Institute, American Mathematical Society, 1993, to appear.

Thinking Machines Corporation, *CM5 Technical Summary*, Thinking Machines Corporation, Cambridge, MA, 1992.

Thinking Machines Corporation, *CMMD Version 3.0 Reference Manual*, Thinking Machines Corporation, Cambridge, MA, 1993a.

Thinking Machines Corporation, *VU Programmer's Handbook - CMOST Version 7.2*, Thinking Machines Corporation, Cambridge, MA, 1993b.

Yepez, J. G. Seeley, and N. Margolus, "Lattice Gas Fluids on Parallel Supercomputers, *submitted to Computers in Physics*, 1994.

APPENDIX A. CODING CONVENTIONS

li,lj	Bounds of a pn's portion of simulation space
Sv	time t lattice memory
SNEXTv	time t + 1 lattice memory
PIPS(i,j,val,addr)	[Put Into Parallel Space] Write (val) into vu memory at lattice site i,j This macro automatically selects the appropriate VU to write to.
GFPS(i,j,addr)	[Get From Parallel Space] Read from vu memory at lattice site i,j into addr. This macro automatically selects the appropriate VU to read from.

APPENDIX B. ARGUMENTS

Recognized Command line arguments to CMCAM:

```
Required:
-nsteps [int]       -> number of steps to run
-report_freq [int] -> display and statistics gathering interval
-pattern_file [fname] -> filename of initial conditions pattern
-rlut_file  [fname]   -> filename of the right handed lut
-llut_file  [fname]   -> filename of the left handed lut
-sys_x [int]          -> system x dimension
-sys_y [int]          -> system y dimension

Optional:
-forcing [float]   -> number of momentum units per time step to add in
                      each direction
-xdisplay     -> if present, an X window is opened and displays the state
                 every report_freq timesteps
-cam_pattern       -> the pattern file is a CAM format pattern file
-colordisp         -> color display
-display_table_file -> file that maps automaton states to
                       8-bit color indices
-palette_file       -> file that maps 8-bit color indices to RGB colors
-send_frames [hostname:directory] -> enables automatic rcp of generated
                                    frames to specified host
```

APPENDIX C. CAM-8 COMPATIBILITY

CMCAM is compatible with the MIT Information Mechanics Group's CAM-8 machine in the following ways.

1. The format of the collision rule lookup table is identical. LUT's produced for the CAM are usable directly by CMCAM.
2. CAM display tables and color palettes are directly usable by CMCAM.
3. CAM initial pattern data is consumable by CMCAM.

Streaming information is encoded in a different manner on each machine and there is no provision for directly transporting code from one machine to the other.

APPENDIX D. OBTAINING AND RUNNING THE CODE

CMCAM source code may be currently obtained from Guy Seeley, email address: seeley@wind.plh.af.mil, phone (617)377-2475.

1. Uncompress and untar the file containing the source and appropriate supporting files.

2. Type **make**. This should compile all the modules and produce an executable known as **lutcm**.

3. Make sure that you are operating on a color X-Window terminal with your **DISPLAY** environment variable appropriately set and access to your own X display enabled.

4. Type **sample_run**. This will run a sample calculation on a 32 node partition of a CM-5. This calculation will display an X-window view of the simulation every 250 time steps.